SwinxsSDK

Documentation

Copyright © 2009 Swinxs BV
The contents of this document is confidential and protected by Dutch law and international treaties. Nothing from this document may be disclosed or copied, by any means, electronic, magnetic or by photocopy, without the prior written permission of the publisher.

Swinxs BV
Oudenhof 14b
4191 NW Geldermalsen
The Netherlands

mail: info@swinxs.com
http://www.swinxs.com

KvK Rivierenland: 17118466
BTW: NL 808347433B01
Rek. nr.: Rabobank 13.06.47.268
IBAN: NL89 RABO 0130 6472 68
Swift Code BIC: RABONL2U

Release history

Version	*Date*	*Author*	*Comments*
0.1	2008-03-01	Mark Ruys	Initial version
1.0	2008-03-31	Mark Ruys	Swinxs 1.0 version
1.1	2008-04-25	Mark Ruys	Added RFID_0 to RFID_9
1.9	2008-06-25	Mark Ruys	Added preliminary references to SwinxsTalk
2.0	2008-08-06	Mark Ruys	Added SwinxsTalk documentation
2.1	2009-01-23	Mark Ruys	Added game.debug.* statements Removed RESIDUE from ADD and SUB in SCT Added predefined constants MINPLAY and MAXPLAY Depreciated RFID_0 to RFID_9 and XS_0 to XS_9 Added DATE, TIME, SND_FILE_BG, SND_TYPE_BG, SND_VOLUME_BG, PLAY_BG, and PLAYING_BG Merged conditions into boolean expressions. Basic arithmetic on constants. Expression allowed at right hand site of compare. Added new syntax for arithmetic and arrays. Enhanced LED subsystem. Moved SwinxsCoreTalk documentation to a separate document. sound.chain() accepts arbitrary expressions as target state. Explanation of the emulator's UI.
2.2	2009-02-04	Mark Ruys	Allow '=' in addition to ':=' as assignment operator.
2.3	2009-02-17	Mark Ruys	Added game.finish.wait and sound.background.fade.
2.4	2009-03-16	Mark Ruys	Added sound.record
2.5	2009-06-29	Mark Ruys	Added led.hsv
2.6	2009-07-10	Mark Ruys	Added comments /* .. */ in SwinxTalk source code. Added sound.jukebox.*, sound.background.fade, button.menu.*, button.volume.*, swinxs.capability
2.7	2009-09-07	Mark Ruys	Added timer.on(<value>) Added random(<lo>, <hi>) Added button.power.*() Added sound.volume, sound.check, sound.exists, and sound.remove
2.8	2009-11-03	Ron Oosterholt	Added game design guidelines
2.9	2009-12-22	Mark Ruys	Added sound.pause.* Changed clock.* firmware revision from 310 to 499
3.0	2010-02-12	Mark Ruys	Added instruction for pausing audio.
3.1	2010-05-11	Mark Ruys	Added led.rfid to show the color of an rfid tag.

Table of contents

1. Summary....4
2. Introduction....5
2.1 Swinxs Usage....5
2.2 Swinxs software architecture....5
2.3 Swinxs Source....6
2.4 SwinxsEmulator....7
3. System description....8
3.1 Sensors, inputs and outputs....9
3.2 SwinxsCoreTalk Virtual machine....10
3.3 Swinxs file system....12
4. SwinxsTalk....16
4.1 Declarations....17
4.2 Finite State Machines....18
4.3 Comments....30
4.4 Include files....30
4.5 Examples....30
5. SwinxsEmulator....32
5.1 Settings....33
5.2 Debugging code....34
5.3 Downloading your game to Swinxs....34
5.4 Uploading your game to the developers site....35
5.5 Shortcut keys....35
5.6 Text-to-speech....36
5.7 Syntax highlighting....36
6. Game development cycle....37
7. Game design guidelines....38

1. Summary

This document describes the Swinxs Software Development Kit (SwinxsSDK). This should help you to develop your own Swinxs applications.
For support, latest versions, etc, see the developers community at http://developers.swinxs.com/.

Intended audience

Target for this documentation are Swinxs developers. The writers have tried to address a non-technical audience also. Unfortunately, it is unavoidable to elaborate on technical details. For some parts of this documentation, it is assumed that the reader has a general (BSc level) technical background.

2. Introduction

Swinxs is a high-tech games console, sustainably designed for active games you can play inside or outdoors. The unique XS-wristband gives you access to a new world to play games, listen to music, experience exciting adventures and make fantasies reality. Connect your Swinxs to your computer and on your own Swinxs page you can, as well as many other things, keep track of your scores and download new games.
Swinxs company encourages the development of new and exciting games for the Swinxs. In order to create new games we provide you with the SwinxsSDK. The software development kit consists of:

1. the SwinxsEmulator and SwinxsTalk compiler,
2. the Swinxs Source, and
3. this documentation.

2.1 Swinxs Usage

Turn on Swinxs using the white button. Swinxs plays an intro. The first you should do is to register all players by holding the player's XS to Swinxs. An XS is a RFID-tag identifying kids. XS stands for 'access'.
Swinxs game play is controlled by RFID-tags and buttons. The buttons have the following meaning:

- Press and hold the white button for a few seconds to turn Swinxs on or off.
- Press the white button shortly to change the volume in three steps.
- Press and hold the red button to enter the game menu. Swinxs will iterate through all games for which enough players are registered. Some games need three or more players, but most need only one or two.
- The green button is used for the happy, positive, game flow. Red vice versa.

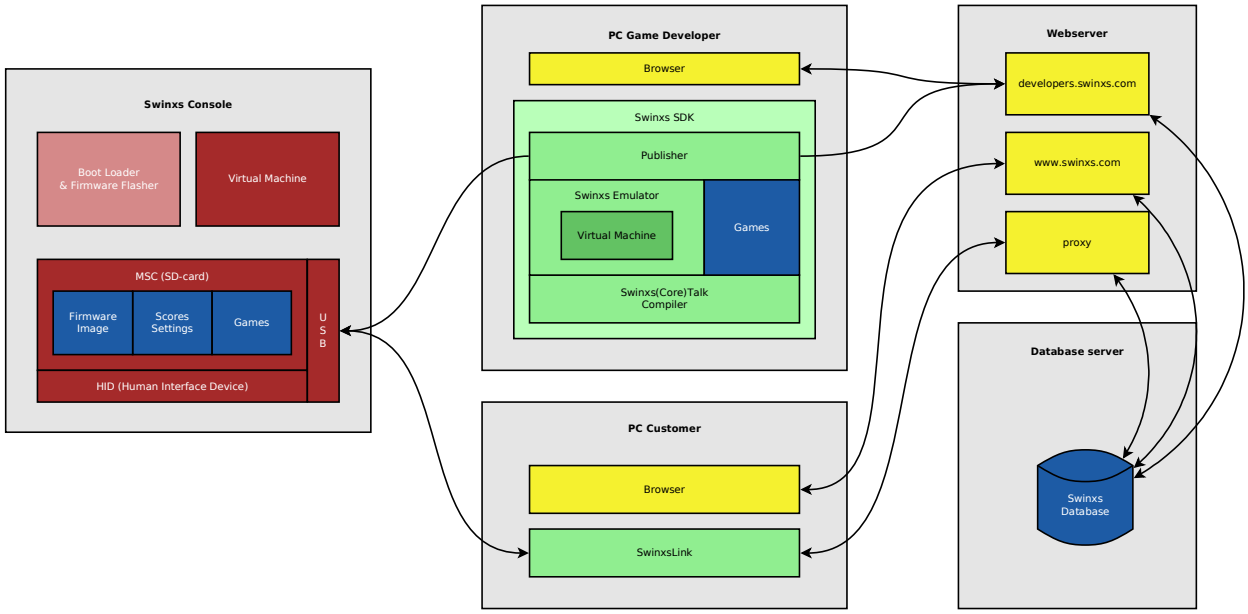
Some games, like Animal Game, needs the Swinxs playing cards. To select such a game, enter the game carousel by pressing the red button for two seconds. The you can select playing card games by showing a playing card game to Swinxs.
See also the chapter “Game design guidelines” at page 38.

2.2 Swinxs software architecture

The Swinxs software architecture consists of the following parts:

1. The Swinxs console
2. The developer PC running the Swinxs SDK
3. The customer PC running SwinxsLink
4. The web- and database server

The purpose of SwinxsLink is to communicate between Swinxs and the swinxs.com website. This document focuses on the Swinxs SDK.



2.3 Swinxs Source

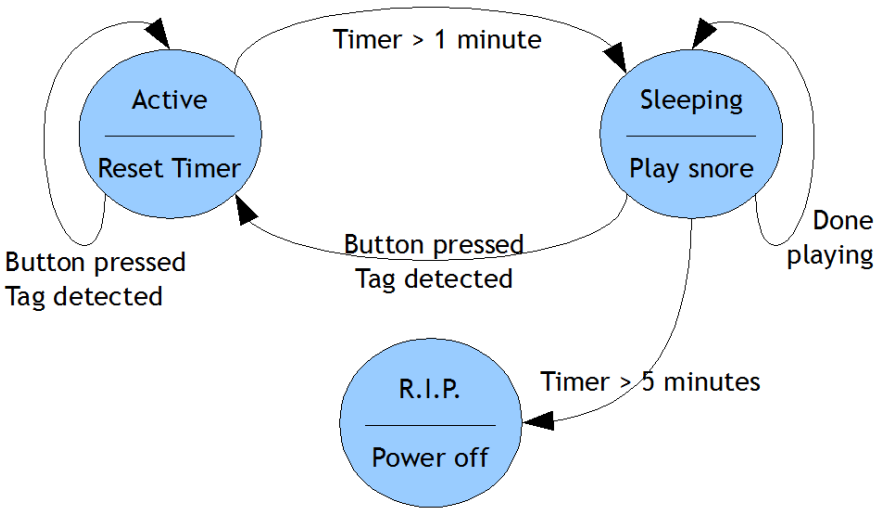
The Swinxs Source itself is divided in audio material and the source code of games. This source code is written in a dedicated language SwinxsCoreTalk or in a higher level language called SwinxsTalk. The audio material consist of uncompressed WAV files. You can find the audio samples on your Swinxs.

SwinxsCoreTalk

SwinxsCoreTalk is a low level language to specify Swinxs games. It's concepts are related to PLC's (Programmable Logic Controllers). A PLC basically maps inputs (XS-wristbands read, buttons, accelerometer) to outputs (LEDs, audio). The mapping can been seen as the program or game specification.
The SwinxsCoreTalk games can be executed by the virtual machine (VM) running the Swinxs.
For more information on SwinxsCoreTalk the reader is referred to the SwinxsCoreTalk documentation.

SwinxsTalk

Because it is rather tedious to develop games in SwinxsCoreTalk, we developed a higher-level language called SwinxsTalk. We expect that anyone with some basic programming skills should be able to get started quickly and develop his or her own games. The syntax of SwinxsTalk is slightly similar to languages like JavaScript.
Basic concept of SwinxsTalk is Finite State Machines (FSM). Here's an example of an FSM:



You see three states: Active, Sleeping and R.I.P. Each state has an entry action like “Reset Timer”. Based on events like “Timer > 1 minute”, a state transition takes place.
The first SwinxsTalk compiler was developed by Richard Schaaf. As of version 0.1.0, the compiler has been open-sourced (GPLv3) and is maintained and extended by Swinxs company itself.

VisualSwinxsTalk

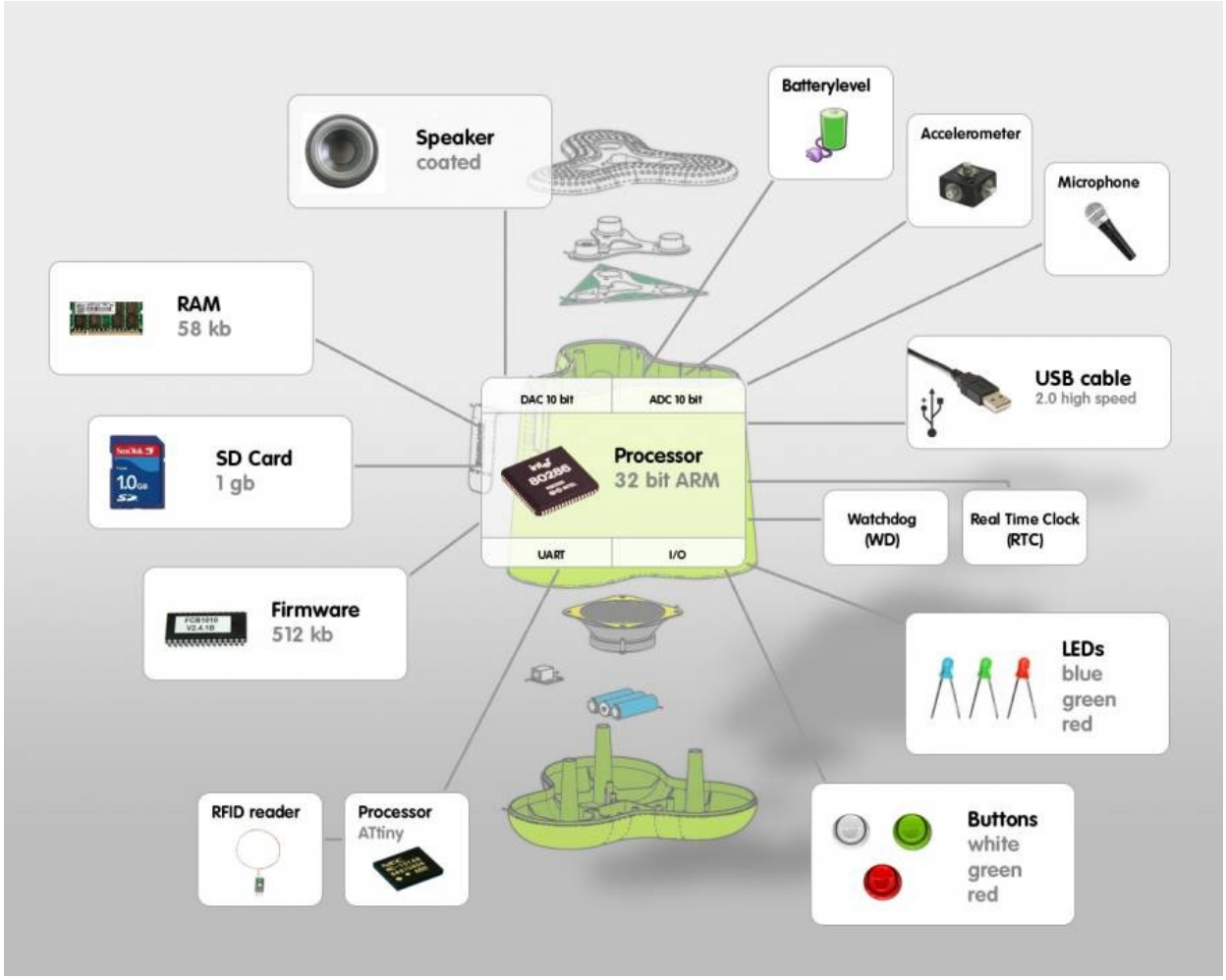
In the not so near future, we expect to design an IDE (integrated developer environment) based on object oriented and visual concepts. This way, it should be far more easy to develop new games. Goal is to enable clever kids to create their own new games.

2.4 SwinxsEmulator

The SwinxsEmulator has two different purposes. Firstly it enables you to develop and debug your Swinxs games. Secondly, it can compile all your games to a binary format which a Swinxs can execute. It will publish your games to your Swinxs, or upload it for you to the developers.swinxs.com site.

3. System description

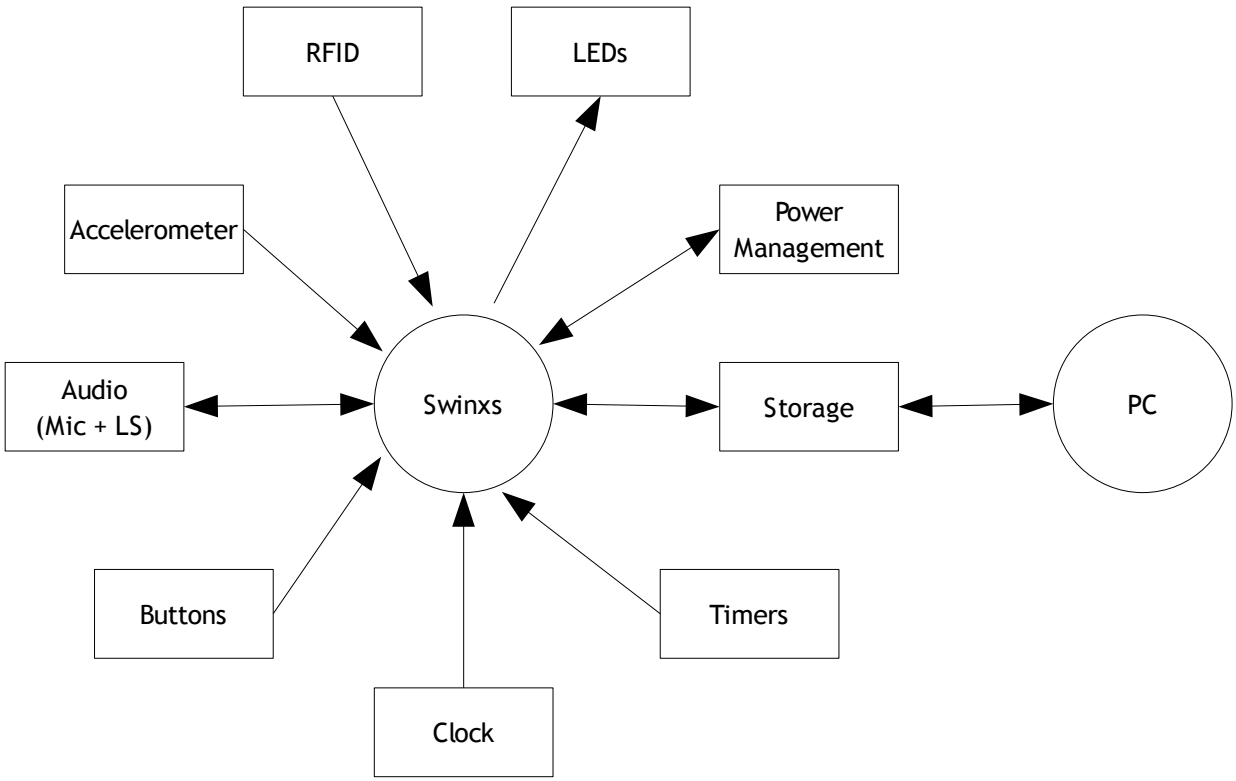
Swinxs is an open gaming platform that acts as a catalyst for social physical game play for 3 to 12 year old children. Swinxs guides children through a set of games.



The Swinxs hardware is a simple device with only three buttons, however more hidden sensors are available. The area enclosed by the three buttons is capable of lighting up over 60 different colors. But the main interfaces of Swinxs are its RFID sensor and sound output device.
Swinxs supports multiple languages (language packs).

3.1 Sensors, inputs and outputs

Swinxs has the following in- and outputs available:



RFID

RFID tags can be detected and read. Each tag has a unique 24-bit number (ID), plus 16 bits to mark special features. Currently we distinguish between XS-tags (bracelets) and playing cards.

Buttons

Swinxs has three buttons (white, green and red). Each button has 4 triggers associated with it:

- button is being hold down
- button has been pressed shortly (less the 500 msec)
- button is pressed long (longer than 500 msec)
- button is pressed and hold down for more than 2 seconds

LEDs

Swinxs has three LEDs blended to a single color. Each LED can be set to 32 intensities, ranging from blacked out to bright white. LEDs can be addressed by RGB or by HSV values.

Accelerometer

You receive events when Swinxs is moved or when Swinxs is shaken. The thresholds which define the sensitivity can be modified through two settings: length and intensity.

Loudspeaker

The audio system is able to play uncompressed wave files. You can play an audio sample, and abort it if you want so. You are notified while audio is being played. During playback, you can adjust the volume. Swinxs can mix audio streams real time.
Typically, Swinxs audio files are uncompressed wave-files sampled at 22 Khz, 16 bit depth, mono.

Microphone

Microphone to detect someone is talking (loud). It's just a trigger.
Also, Swinxs is able to perform voice recordings which are saved to Swinxs' storage.

Timers

Swinxs has 13 freely available timers running at a resolution of 100 msec. Each timer can be initialized, started, stopped and read out. Maximum value is 65535, which holds roughly 100 minutes.

Clock

As long as the Swinxs battery does not run out of power, Swinxs maintains a real time clock.

Power control

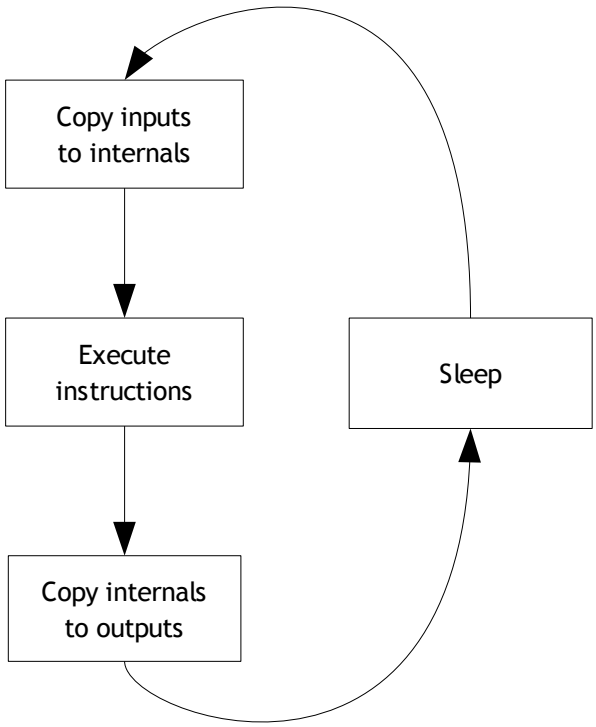
When the Swinxs reaches the low power condition, it is detected as an input. You can power off Swinxs from within your application.

Storage

Integer application variables can be saved to persistent memory and loaded back into Swinxs memory on a per-game basis.
When Swinxs is attached to a PC, the persistent memory is available as a USB flash drive. This drive contains all applications (games), game meta-information, audio samples, scores, saved Swinxs memory, etc.

3.2 SwinxsCoreTalk Virtual machine

Originally, Swinxs games are coded in a specially designed language called SwinxsCoreTalk. SwinxsCoreTalk code is interpreted by a virtual machine. This virtual machine is implemented as an finite state machine.



Control cycle

The virtual machine basically reads all inputs, process a set of instructions (the program) and write all outputs. Completing a full cycle of the above finite state machine is referred to as a control cycle. The instructions comprehend the game definition. It is the program which is executed every control cycle.
When an instruction sets an output, it is actually only performed at the “copy internals to outputs” phase. So when you have two instructions which sets or resets the same output within a single control cycle, the first instruction is effectively ignored. For example, there will be no audio heared if you start playing a sample and then stop playing samples within the same control cycle. It might take some experience to get used to this programming paradigm.
The sleep delay is typical 40 msec but may fluctuate[1]. When Swinxs is busy, control cycles may even be dropped.

Accumulator and register

The core of the virtual machine consist of binary accumulator and a 16-bit word register. A word is a unsigned integer from 0 up to 65535. You could regard the accumulator and register as the main memory of a Swinxs.

Variables

The virtual machine further supports 256 binary variables, some of which are virtual I/Os, usually connected to real I/Os and 256 word variables, some of which have a special purpose.

[1]This may change in the future. Use timers when you need an accurate notion of time.

To avoid spurious signals on the outputs these are set after completion of the control cycle. For similar reasons the inputs are copied to internal variables before the program executes.

Instruction memory

Part of the virtual machine is also a 6000 byte instruction memory for game specifications. A single SwinxsCoreTalk instruction typically takes two bytes.

3.3 Swinxs file system

The Swinxs hooks up to a PC by simulating an USB-memory stick and appears to the PC user as a removable disk. Programming Swinxs basically involves nothing more than transferring files to the device. Other then that Swinxs assumes a file structure as shown below:

/boot.bin				*Boot game definition*
/00000.wav				*Sound file(s) for boot game*
/eng/				*English games and audio*
	sys/	00000.wav		*System sound file(s)*
	bracelets/	BL0.wav		*Names of XS*
		00008721.wav		*Personalized XS audio*
	numbers/	00000.wav		*Cardinal numbers*
		00001th.wav		*Ordinal numbers*
	games/	00000/	00000.bin	*Game selection game*
			00000.wav	*Game selection audio file(s)*
		00001/	00000.wav	*Jukebox songs*
		00011/	game.prp	*Game properties*
			00000.bin	*Game number 11*
			00000.wav	*Title of game*
			00001.wav	*Other audio file(s)*
		:		
		:		
/nld/				*Dutch games*
:				
:				

Filenames

Most filenames consist of 5 digits, followed by some optional text, a dot and the extension. The optional text is ignored. The 5 digits must be in the range of 0 to 65535 and correspond with word values used by the games.
The following extensions are used:

- WAV: uncompressed wave audio
- SCT: SwinxsCoreTalk files with game definitions
- ST: SwinxsTalk files with game definitions
- BIN: compiled SwinxsCoreTalk files
- PRP: property files (plain VAR=VALUE text files)
- XML: XML files like scores

Personalized XS audio files always consist of 8 digits representing their 24-bit unique ID.

Root

In the root of the Swinxs memory-stick, you find the file boot.bin. This contains the boot game which runs when the Swinxs is powered on. This game can use language independent audio files located in the root.
Next you see a directory per supported language. This directory must be a three letter code, typically based on ISO 639-3.

Multiple languages

A language pack consists of:

- System audio files for generic use
- XS names
- Numbers and ordinals
- Game selection game
- Jukebox audio directory
- Several game directories

Games

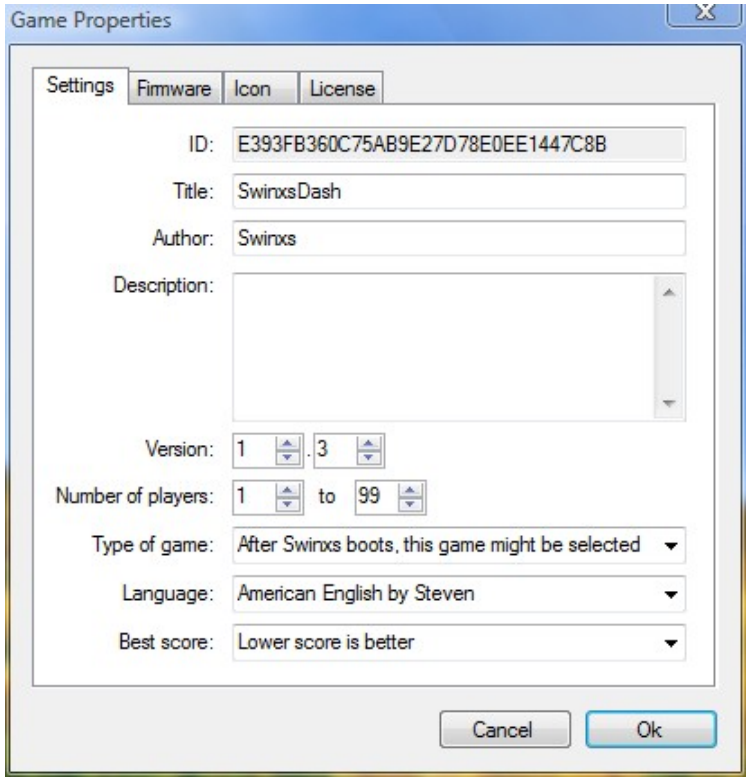
Each game needs at least three files:

- game.prp which specify several game properties
- 00000.wav for the game title (used in the game selection menu)
- 00000.bin which contains the game definition

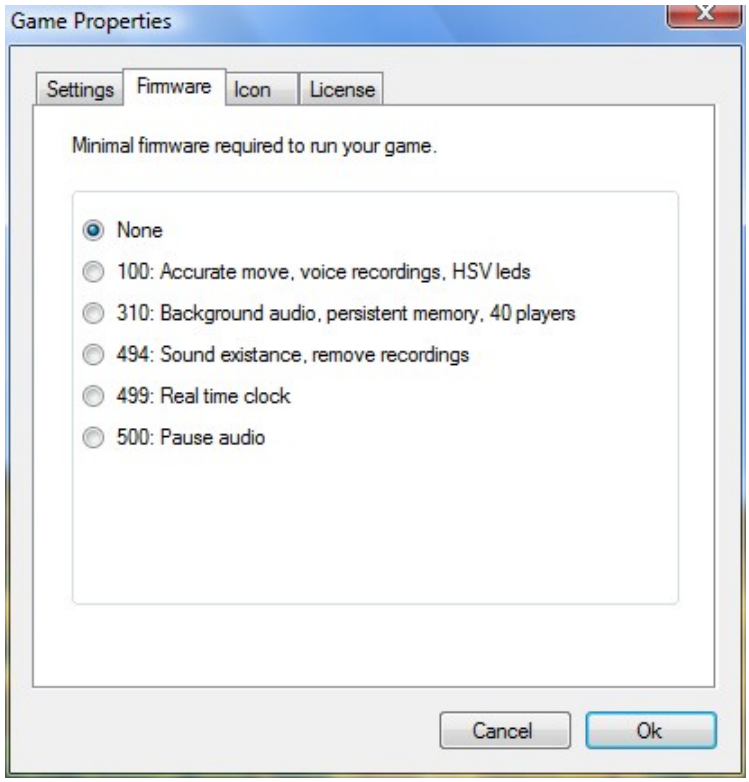
Furthermore, it can contain several game specific audio files and optionally several game definitions overlays (*.bin files).
Game definitions are binary objects produced by the SwinxsEmulator. The SwinxsEmulator compiles games written in Swinxs(Core)Talk to binary code which can be executed by the Swinxs virtual machine.

Game properties

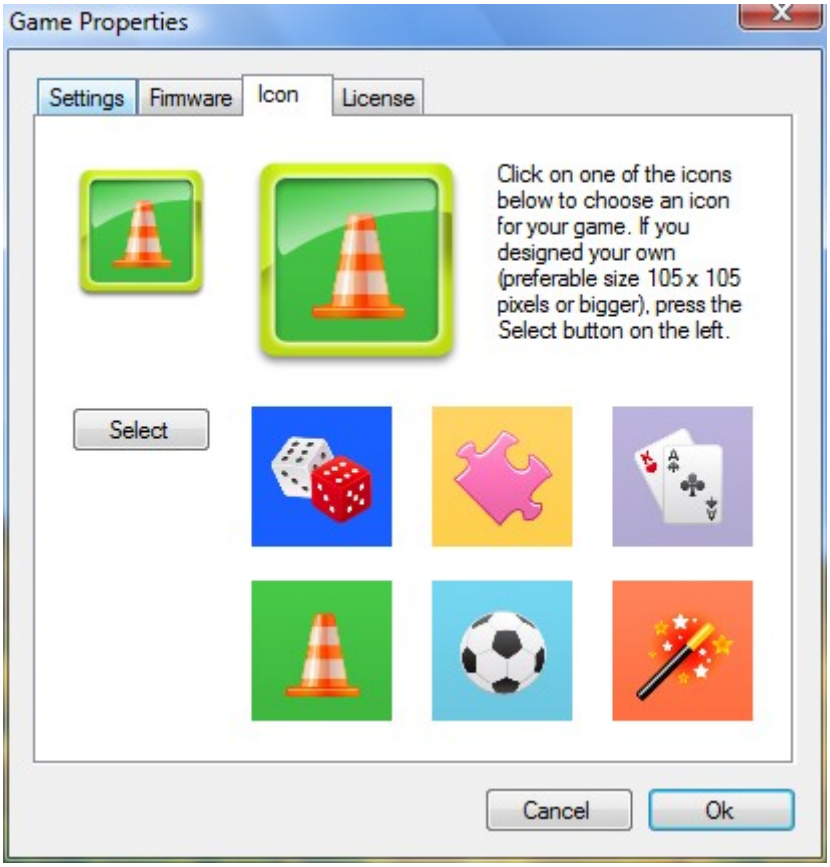
Each game must have a game property file (game.prp). This is a text file defining all game settings like how many players the game support, whether its needs a minimal firmware version, if it will be a paid game, etc. Use the SwinxsEmulator to fill in the values:



If your game requires a minimal firmware release, use the firmware tab:



Use the tab “Icons” to select one of the predefined icons, or choose your own:


Game Properties
Settings
Firmware
Icon
License
Click on one of the icons below to choose an icon for your game. If you designed your own (preferable size 105 x 105 pixels or bigger), press the Select button on the left.
Select
Cancel
Ok

4. SwinxsTalk

Before we describe the SwinxsTalk language in detail, we first give an example. It's an implementation of the state machine shown in section 2:

```
variables
{
    const ACTIVE   = 100;
    const SLEEPING = 200;
}

machine
{
    startup
    {
          rfid.on;                  // Enable RFID detection
          timer.on(0, 0);           // Start timer 0 at 0 seconds
          state.move(ACTIVE);
    }

    state(ACTIVE)
    {
          if ( button.green.pulse || rfid.detected )
          {
               timer.set(0, 0);  // Reset timer 0 to 0
          }

          if ( timer.get(0) > 600 ) // We measure time in tenths of secs
          {
               state.move(SLEEPING);
          }
    }

    state(SLEEPING)
    {
          sound.play("SYSTEM", 39); // Plays system sample 00039.wav

          if ( button.green.pulse || rfid.detected )
          {
               sound.off;
               state.move(ACTIVE);
          }

          if ( timer.get(0) > 3000 )
          {
               power.off;
          }
    }
}
```

So the pattern is: you have a variable and constant declaration followed by a finite state machine specification. The state machine has an entry state (startup) and can jump to other states when needed (state.move).

4.1 Declarations

Your program may contain one or more declaration blocks. A declaration block consists of:

1. Boolean variable declarations. A boolean variable can hold either true or false.
2. Word variable declarations. A word variable can contain a value from 0 to 65535, so it's unsigned.
3. Array of word variable declaration. Array size should be fixed.
4. Constant declarations. Basic integer arithmetic (+, -, *, /, and %) is allowed.
5. Machine declarations. When you want the run more then one state machine in parallel, declare the additional state machines in a variable declaration block.

Swinxs has only a very limited storage for boolean and word/array variables: about 150 of each type depending on the complexity of your program. When a state machine starts, no variables are initialized.

SwinxsTalk keywords are case insensitive. However, identifiers like variables and constants are case sensitive.

An example of a variable declaration block:

```
variables
{
    const MAX_SCORES = MAXPLAY * 2;

    word player;
    word scores[MAX_SCORES];
    bool playerIsRunning;

    machine secondMachine;
}
```

Predefined constants

SwinxsTalk has some predefined constants which may be convenient for some games.

Name	*Description*
DEBUG	When debug mode is enabled in the emulator, holds “1”. Otherwise “0”.
LANGPACKID	Language pack ID as defined in game.prp. May be used to write language depended code.
MINPLAY	The minimum number of players needed to play this game (as defined in game.prp).
MAXPLAY	The maximum number of players allowed to play this game (as defined in game.prp).

For example, declare an array to hold half of the number of players, declare:

```
variables
{
    word team1[(MAXPLAY + 1) / 2];
}
```

4.2 Finite State Machines

A finite state machine definition consist of a series of states. You can define multiple state machines in your game. This can be convenient to write a dedicated state machine for a specific task. Each state machine you define will run a parallel.

```
machine
{
    …
}

machine(secondMachine)
{
    …
}
```

A state machine should have a label. But if you omit the label, the label “DEFAULT” is used.

States

A state machine has three types of states:

1. Labeled state. These states are ordinary states identified by an integer ranging from 1 up to 60000 (exclusive).
2. Start-up state. Typically, every state machine will have an entry state, called the start-up state. This is where the execution of your state machine begins.
3. Always-state. As the name suggests, an always-state is always executed, regardless the state of your machine.

As long as a state does not change, all code within the state-block is run and rerun approximately 25 times a second. A state may have an “enter action”. Code defined in an enter block is executed only once when entering the state from another state. Example:

```
machine
{
    startup
    {
        rfid.on;
    }

    state(ACTIVE)
    {
        enter
        {
            i = 0;
        }

        if ( i < 10 )
        {
            i++;
        }
    }

    always
    {
        ...
```

```
    }
}
```

Of course, you could also initialize the variable i in the startup state, that's up to you.

Statements

First we introduce a few general statements.

Syntax	*Meaning*
{ <statements> }	Groups semicolon separated statements together
if (<expr>) <statement>	Execute a <statement> if the boolean <expr> is true.

if (<expr>)
  <statement>
else
  <statement>
	The else-part is optionally.

<var> = <expr>	Assign an expression to a word or boolean variable 'var'.
<var>[<expr>] = <expr>	Assign an expression to an element of the array 'var'. Indexes should run from 0 up to, but not including, the size of the array.
<var>++	Increment word <variable> by one.
<var>--	Decrement word <variable> by one.

We also have dedicated statements for the various subsystems. These are described later on.

Numeric expressions

These are the basic numeric expressions. All arithmetic is done modulo 65536, so all numbers are unsigned and range from 0 to 65535.

Syntax	*Meaning*
<constant>	An integer or a constant defined in a declaration-block.
<var>	Retrieves the value of <variable>.
<var>[<expr>]	Retrieves from the array <var> the value indexed by <expr>. Indexes should run from 0 up to, but not including, the size of the array.
<expr> + <expr>	Addition.
<expr> - <expr>	Subtraction.
- <expr>	Negation. Note that as words are unsigned numbers, negation is equal to 65536 - <expr>.
<expr> * <expr>	Multiplication.
<expr> / <expr>	Integer division. Division by zero equals zero.
<expr> % <expr>	Modulo. Modulo zero returns zero.
<test> ? <expr> : <expr>	If <test> is true, it equals the first <expr>, otherwise the second.

min(<expr>, <expr>)	Minimum.
max(<expr>, <expr>)	Maximum.
random(<expr>)	Returns a random value between 0 and (<expr> - 1) inclusive.
random(<lo>, <hi>)	Returns a random value between the expressions <lo> and <hi>.

Boolean expressions

An boolean expression defined as follows.

Syntax	*Meaning*
<constant>	True or false.
<var>	Retrieve the value of a boolean variable.
<expr> && <expr>	True only if both arguments are true.
<expr> || <expr>	True only if one or both of the arguments are true.
! <expr>	True only if the argument is not true.
<test> ? <expr> : <expr>	If <test> is true, it equals the first <expr>, otherwise the second.
<expr> == <expr>	Equality test on a variable.
<expr> != <expr>	Inequality test on a variable.
<expr> < <expr>	Test whether a variable is lesser than an <expr>.
<expr> <= <expr>	Test whether a variable is lesser or equal to an <expr>.
<expr> > <expr>	Test whether a variable is bigger than an <expr>.
<expr> >= <expr>	Test whether a variable is bigger or equal to an <expr>.

State subsystem

Syntax	*Type*	*Meaning*
state.move(<expr>)	statement	The state machine will jump to state <expr> at the next cycle.
machine.state.move(<var>, <expr>)	statement	The state machine <var> will jump to state <expr> at the next cycle.

Timer subsystem

You have 13 timers to your possession, numbered from 0 to 12. All timers have a precision of 100 milliseconds. So using a single timer you can measure 0.1 seconds up to 100 minutes.

Syntax	*Type*	*Meaning*
timer.on(<value>, <expr>)	statement	Start running timer <value> from <expr> and up.
timer.set(<value>, <expr>)	statement	Set timer <value> to <expr>.
timer.off(<value>)	statement	Pause timer <value>.
timer.on(<value>)	statement	Resume timer <value>.
timer.get(<value>)	expression	Retrieve the current time of timer <value>.
timer.running(<value>)	boolean	True while timer <value> is ticking.

<value> should be a constant or a word variable.
Note that while a timer equals 0, the expression timer.running() is always false. This way you can check when a timer looped. So if you test whether a timer is running, you should avoid to initialize the timer with 0.

Clock subsystem

Swinxs has a real time clock which is synchronized to the user's PC by SwinxsLink when Swinxs is docked.

Syntax	*Type*	*Meaning*
clock.gettime(<hour>, <minute>)	statement	Fills the word-vars with the current hour (0 to 23), and minutes (0 to 59) respectively.
clock.getdate(<year>, <month>, <day>, <dayofweek>)	statement	Fills the word-vars with the current year (2008 and up), month (1 to 12), day (1 to 31) and the day of the week (0 to 6, where 0 stands for Sunday) respectively.

To check if the firmware supports the real time clock, use the boolean expression swinxs.capability("CLOCK").

RFID subsystem

You have to distinguish between RFID tags and Xses. A tag is any object containing a Swinxs readable RFID token in it. Every tag has a unique 24 bit number assigned.
An accessory tag is a special tag with an RFID token inside.
An XS is a registered bracelet. Before a game is started, players are asked to hold their XS above Swinxs. These Xses are registered and you can refer to them in your program. We also have a special play command to call out an XS his name.

Syntax	*Type*	*Meaning*
rfid.on	statement	Turn on the detection of tags.
rfid.off	statement	Turn RFID detection off, e.g. to save energy.
rfid.detected	boolean	True for one cycle when a tag is detected. Fills rfid.last.lo and rfid.last.hi.
rfid.last.lo	expression	Return the RFID number modulo 65350.
rfid.last.hi	expression	Return the RFID number divided by 65350.
rfid.accessory.detected	boolean	True for one cycle when a accessory tag is detected.
rfid.xs.register.on	statement	Start a new registration procedure. Typically used in the game selection program.
rfid.xs.register.off	statement	Terminates the registration procedure.
rfid.xs.count	expression	Returns the number of registered Xses.
rfid.xs.detected	boolean	True (for one cycle) when a registered XS is detected. Stores the index of the XS in rfid.xs.last.
rfid.xs.last	expression	Returns the index of the current XS.

The RFID number modulo 10 determines the color of the XS. The colors are defined as follows:

Last digit	*Color*	*Last digit*	*Color*
0	Orange	*5*	White
1	Yellow	*6*	Red
2	Green	*7*	Violet
3	Cyan	*8*	Pink
4	Black	*9*	Grey

Use the swinxs.capability("MAX_PLAYERS") expression to determine how many players the firmware supports. All Swinxses can handle at least 10 players pro game.

Score subsystem

Syntax	*Type*	*Meaning*
score.set(<XS>, <expr>)	statement	Stores expression <XS> in rfid.xs.last and <expr> in score.last. Saves the next cycle the score for this XS in the internal score table.
score.get(<expr>)	expression	Returns the score of the XS <expr>. *Valid only for <expr> from 0 up to 9 (e.g. the first 10 players).*
score.get(<expr>)	statement	Stores XS <expr> in rfid.xs.last and, at the next cycle, fills score.last (from the score table), rfid.last.lo and rfid.last.hi [2].
score.last	expression	Returns the score of rfid.xs.last.
score.saveall	statement	Write the score table of all registered XSes to the persistent memory. When the user connects his/her Swinxs the the PC, these scores are sent to swinxs.com.

Sound subsystem

Swinxs can play sounds from various sources:

- SOUND: audio files from your own directory
- XS: the name (color) of a registered bracelet (ranging from 0 to rfid.xs.count - 1)
- NUMBER: a number from 0 up to and including 100, plus 1000
- ORDINAL: an ordinal from 1 to 20
- SYSTEM: a system wide available sound (see the Excel-file sound samples.xls for a list of all available sound samples in the 'sys' directory)
- JUKEBOX: play sample from the jukebox directory
- SILENCE: be quiet for <expr> tenths of seconds (only valid for the sound.chain statement)

In the table below, <type> refers to one of the keywords as a quoted string (e.g. "SOUND", etc).

[2]Note that by score.get() sets rfid.last.lo and rfid.last.hi only for firmware releases that support 40 or more players. Use swinxs.capability("MAX_PLAYERS") to check.

Syntax	*Type*	*Meaning*
sound.play(<type>, <expr>)	statement	Plays audio file <expr> of type <type>.
sound.chain(<expr>, <chain>)	statement	Plays a sequence of foreground sound samples. After starting the last sample, jumps to state <expr>. This command will commence as soon as no sample is playing.
sound.off	statement	Aborts playing a sample.
sound.playing	boolean	True while sound is playing.
sound.volume	expression	Gets the current foreground volume.
sound.volume(<expr>)	statement	Sets the foreground volume (from 45 to 100).
sound.visual.on	statement	Normally, when Swinxs speaks (plays audio), the LED pulses green. You enable this with sound.visual.on.
sound.visual.off	statement	Stops pulsing the LED while Swinxs speaks, e.g. if you want to program the LED yourself.
sound.background.play (<type>, <expr>)	statement	Plays background audio file <expr> of type <type>. Background sounds are repeated indefinitely until you abort it by calling sound.background.off.
sound.background.off	statement	Aborts playing a background sample.
sound.background.playing	boolean	True while background sound is playing.
sound.background.volume (<expr_bg>, <expr_fg>)	statement	Sets the background audio volume. If foreground audio is playing, use <expr_fg>, otherwise <expr_bg>. Ranges from 0 (silence) to 100 (same level as current foreground sound).
sound.background.fade (<expr_bg>, <expr_fg>, <step>)	statement	Gradually change background volume settings. The background sound will abort when faded to a volume of 0. Each control cycle the volume gets in- or decremented by <step>. Typically you'll apply a step between 1 (slow) and 10 (fast).
sound.check(<type>, <expr>)	statement	Checks whether the audio file <expr> of <type> exists. You must test sound.exists the next control cycle.
sound.exists	boolean	True if the file specified by sound.check() exists.

The audio sound system as a few subtleties:

1. If Swinxs is playing some sound file, and you issue a sound.play command, it is ignored. You have to wait until sound.playing gets false, and only then set sound.play.
2. If Swinxs is playing some sound file, and you issue a sound.chain command, the command will be executed as soon as the current sample is finished. This makes the sound.chain command very convenient to use, even for a single sample. A <chain> consist of a comma separated list of <type>, <expr>, just like the sound.play command.
3. When a audio sample is finished, the sample is repeated the next cycle. It is your task to check the sound.playing status to play a new sound, or to stop playing by setting sound.off.

4. When you abort the sound subsystem by calling sound.off, it may take a few cycles before sound.playing gets low. Typically you should test sound.playing getting low before you issue a new sound.play command.
5. Mixing of back- and foreground audio has three limitations:
   1. Only supported by firmware revisions for which swinxs.capability("BACKGROUND_AUDIO") is set;
   2. Both audio streams must be of the same sample rate;
   3. Only background types SOUND and SYSTEM are allowed.
6. You may use sound.play(<file>, <expr>) and sound.check(<file>, <expr>) the same control cylce. But only if the <file> and <expr> are identical, as these share the same underlying SwinxsCoreTalk variables.
   To check if the firmware supports the checking on file existence, use the boolean expression swinxs.capability("SOUND_EXISTS").
   When you check the existence of a audio file of type XS, only personalized samples are considered. These samples are typically the result of a voice recording.

Pausing audio

The following instructions are available for pausing audio:

sound.playing.pause	statement	Pauses playing (or recording) audio.
sound.playing.resume	statement	Resume playing (or recording) audio.
sound.playing.paused	boolean	True if foreground audio is paused.
sound.background.playing.pause	statement	Pauses playing (or recording) audio.
sound.background.playing.resume	statement	Resume playing (or recording) audio.
sound.background.playing.paused	boolean	True if background audio is paused.

Due to a firmware limitation, you should not issue a sound.off when paused: first resume the audio to abort it the next cycle. Also note that both fore and background audio is affected. In a future firmware release this might change.
To check if the firmware supports the pausing audio, use the boolean expression swinxs.capability("PAUSE_AUDIO").

Jukebox subsystem

Swinxs supports nine jukebox directories: /xxx/games/00001 upto /xxx/games/00009[3]. You can use these directories to play (background) sounds within your game.

Syntax	*Type*	*Meaning*
sound.jukebox.shuffle.on (<directory>)	statement	Initializes the jukebox using <directory>. Songs are shuffled.
sound.jukebox.sort.on (<directory>)	statement	Initializes the jukebox using <directory>. Songs are sorted by number.

[3]SwinxsLink currently supports only the first jukebox directory.

Syntax	*Type*	*Meaning*
sound.jukebox.next	statement	Moves to the next song.
sound.jukebox.previous	statement	Moves to the previous song.
sound.jukebox.count	expression	Number of songs in the jukebox directory.
sound.jukebox.off	statement	Disables the current jukebox. The next control cycle you could open another jukebox.

Use sound.play("JUKEBOX", <directory>) to play the current song from the jukebox. Make sure you issue this command when sound.playing is false!
The jukebox can also be used with sound.chain() and sound.background.play().
See games 00013 or 00014 for examples how to use the jukebox.

Voice recording subsystem

Swinxs can detect noise and make voice recordings. Also Swinxs can personalize XS-tags by making a voice recording of a player's own name and attach the recording automatically to the XS.

Syntax	*Type*	*Meaning*
sound.record.on(<expr>)	statement	Start a voice recording of a sound sample. <expr> holds the id of the sample in the game directory.
sound.record.add(<expr>)	statement	Start a voice recording of a sound sample. The first free (non used) sample from <expr> or up will be used as the sample id.
sound.record.xs(<expr>)	statement	Start a voice recording for the registered XS <expr> (0 to number of registered players).
sound.record.off	statement	Finish the sound recording.
sound.detected	boolean	True when noise is detected (kids yelling/clapping their hands).
sound.remove(<expr>)	statement	Removes sample <expr>.
sound.remove.xs(<expr>)	statement	Removes a voice recording for the registered XS <expr>.

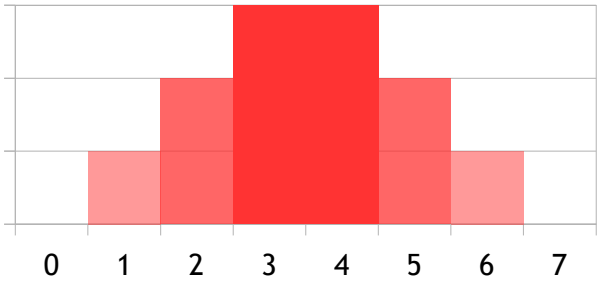
Notes:

- To check if the firmware supports voice recordings, use the boolean expression swinxs.capability("RECORD"). To check if the firmware supports the removing recordings, use the boolean expression swinxs.capability("REMOVE_RECORDING").
- Make sure Swinxs is not playing any audio while making a voice recording.
- Direct the players to hold Swinxs as a microphone. As Swinxs has no automatic gain control, players should be in a short range of Swinxs (approx. 20 cm/10 inch) and should try to use a loud and clear voice.
- For an XS-recording, the name of the particular XS will be the sample just recorded. Each time Swinxs calls this XS, the recording will be used. To use the original sample again, the player should dock his/her Swinxs and remove the personalized XS via the site.
- Recording of sound samples are stored in the game directory as <nnnnn>REC.WAV files where <nnnnn> is the five digit number identifying the sample.

LED subsystem

Syntax	*Type*	*Meaning*
led.hsv(<h>, <s>, <v>)	statement	Set the LED to hue (<h> % 360), saturation (<s> % 101), and value (<v> % 101).
led.color(<color>, <int>)	statement	Set the LED to <color> and intensity <int>. The intensity should be “HIGH”, “MEDIUM”, or “LOW”.
led.rfid(<expr>, <int>)	statement	<expr> should be an RFID number modulo 65350 and <int> should be an intensity specification.
led.pulse(<color>, <speed>)	statement	Make the LED pulse in <color> with a duration of <speed>.
led.off	statement	Makes the LED black (turn off).

<color> should be “BLUE”, “GREEN”, “RED”, “LIME”, “CYAN”, “VIOLET”, “WHITE”, “PINK”, “ORANGE”, “YELLOW”, “BLACK”, or “GREY”.
Using the pulse subsystem you make the LEDs pulse in a specific color. A pulse consists of 6 steps (phases) and uses three different colors (intensities):



A pulse takes roughly 250 msec times the value you store in <speed>.
To check if the firmware supports HSV LEDS, use the boolean expression swinxs.capability("HSV_LEDS").

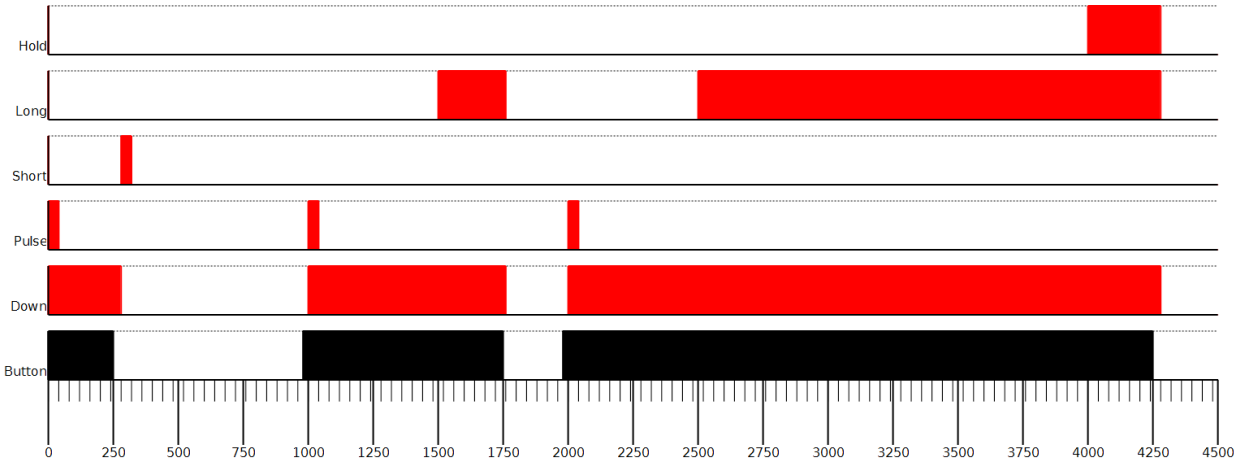
Accelerometer subsystem

Swinxs can detect two kinds of movements: a short move and a long shake.

Syntax	*Type*	*Meaning*
accel.move.detected	boolean	True when Swinxs is (slightly) moved.
accel.shake.detected	boolean	True when Swinxs is shaken.
accel.move.settings(<threshold>, <count>)	expression	Threshold (default 10) and count (default 4) settings defines sensitivity move events.
accel.shake.settings(<threshold>, <count>)	expression	Threshold (default 20) and count (default 10) settings defines sensitivity shake events.

Button subsystem

Swinxs has three buttons: green, red and white. The following chronogram visualizes the timing of various events:



Times are in milliseconds. The bottom row button visualizes the physical button presses. The rows "down", "short", "long", and "hold" corresponds with the virtual inputs below. the row "pulse" is only available for the green button.

Syntax	*Type*	*Meaning*
button.green.down	boolean	True while the green button is being pressed.
button.green.pulse	boolean	True for a single cycle if the green button is pressed.
button.green.short	boolean	True for a single cycle if the green button is pressed shortly.
button.green.long	boolean	True while the green button is pressed for more than half a second.
button.red.hold	boolean	True while the red button is pressed for more than two seconds.
button.red.down	boolean	True while the red button is being pressed.
button.red.short	boolean	True for a single cycle if the red button is pressed shortly.
button.red.long	boolean	True while the red button is pressed for more than half a second.
button.red.hold	boolean	True while the red button is pressed for more then two seconds.
button.white.down	boolean	True while the white button is being pressed.
button.white.short	boolean	True for a single cycle if the white button is pressed shortly.
button.white.long	boolean	True while the white button is pressed for more than half a second.
button.white.hold	boolean	True while the white button is pressed for more than two seconds.
button.volume.on	statement	Enables changing the volume via the white-short button.
button.volume.off	statement	Disables changing the volume via the white-short button.
button.power.on	statement	Enables turning off Swinxs via the white-short button.
button.power.off	statement	Prevent turning of Swinxs via the white-long button.
button.menu.on	statement	Enables the game menu via the red-long button.
button.menu.off	statement	Disables the game menu via the red-long button.

Shuffle subsystem

Syntax	*Type*	*Meaning*
shuffle.on(<expr>)	statement	Initializes the shuffle at the next cycle with a list of length <expr>. Each item contains a unique random value between 0 and <expr> - 1.
shuffle.get	expression	The next item in the shuffle list. Only the next cycle, you will get a new shuffle item.
shuffle.rewind	statement	Rewinds the shuffle list to the beginning.
shuffle.off	statement	You *must* turn the shuffle off for one cycle before you can

Syntax	*Type*	*Meaning*
		start a new one using shuffle.on(..).
shuffle.end	boolean	True when the last element of the shuffle list is read.

Because of limited shared Swinxs memory, the length of the shuffle list *plus* the number of menu items, must be below 180. This memory is freed only when you reinitialize the menu. The menu subsystem is not yet available in SwinxsTalk.
Typical shuffle code looks like this:

```
startup
{
    shuffle.on(rfid.xs.count);
    …
}

state(ST_SELECT_PLAYER)
{
    player = shuffle.get;
    …
}

state(ST_NEXT_PLAYER)
{
    if ( shuffle.end )
    {
        // End if game
        …
    }
}
```

Game subsystem

Syntax	*Type*	*Meaning*
game.start(<expr>)	statement	Load and run a new game file, replacing the current one.
game.finish.score	statement	Exit your game by telling the scores (which are also saved to disk), and then:
game.finish.intro	statement	Exit your game by congratulate the players, and then:
game.finish.button	statement	Exit your game by telling to press the green or red button, and then:
game.finish.wait	statement	Exit your game by waiting for the green (play your game again) or the red button (leave your game and play another).
game.debug.log(<exprr>)	statement	Show the value of <expr> in the SwinxsEmulator status log.
game.debug.break	statement	Within the SwinxsEmulator, stops executing your game and enters step-mode.

You can use game.start() to split your program in multiple parts when your game does not fit in Swinxs memory.Take a look at the game Swinxs Relay (00047) for an example how to implement this.

Memory subsystem

Syntax	*Type*	*Meaning*
memory.word.saveall	statement	Save all word variables and arrays to persistent memory.
memory.word.loadall	statement	Load a previously saved memory dump.

To check if the firmware supports persistent memory, use the boolean expression swinxs.capability("PERSISTENT_WMEN").

Power subsystem

Syntax	*Type*	*Meaning*
power.off	statement	Turn off Swinxs.
power.sleep.timeout (<expr>)	statement	Set the timeout after which Swinxs falls asleep (in tenths of seconds).
power.battery.low	boolean	True when Swinxs is on low power.

Swinxs subsystem

Syntax	*Type*	*Meaning*
swinxs.sct(<string>)	statement boolean expression	Embed the string as SwinxsCoreTalk code, e.g. sct("MVI 10").
swinxs.capability(<string>)	boolean expression	Test whether the installed firmware supports a feature.

This is the list of features you can test for:

Feature	*Firmware*	*Type*	*Description*
ACCURATE_MOVE	100	boolean	More accurate move/shake detection.
RECORDING	100	boolean	Voice recordings.
HSV_LEDS	100	boolean	Allow LEDs to be set by HSV.
BACKGROUND_AUDIO	310	boolean	Real time mixing of background audio.
PERSISTENT_WMEM	310	boolean	Persistent memory for word variables.
MAX_PLAYERS		expression	From firmware version 310, Swinxs supports 40 players pro game. Earlier Swinxses support 10 players.
SOUND_EXISTS	494	boolean	Can check for existence sound files.
REMOVE_RECORDING	494	boolean	Can remove voice recordings.
CLOCK	499	boolean	Real time clock.
PAUSE_AUDIO	500	boolean	Can pause fore- and/or background audio

Swinxs 1 has firmware revision 35 and is not upgradeable. Typically Swinxs 2 users will run your game on the latest firmware version available as we try to push updates as soon as such a user docks his/her Swinxs.

4.3 Comments

Any text between /* and */ is ignored. Comments may span multiple lines. Also, text from // is ignored up to the end of the line.

4.4 Include files

Use the “append” keyword to concatenate multiple SwinxsTalk source files:

```
append “extra.st”
```

The contents of the file “extra.st” are appended to the end of the current source file.

4.5 Examples

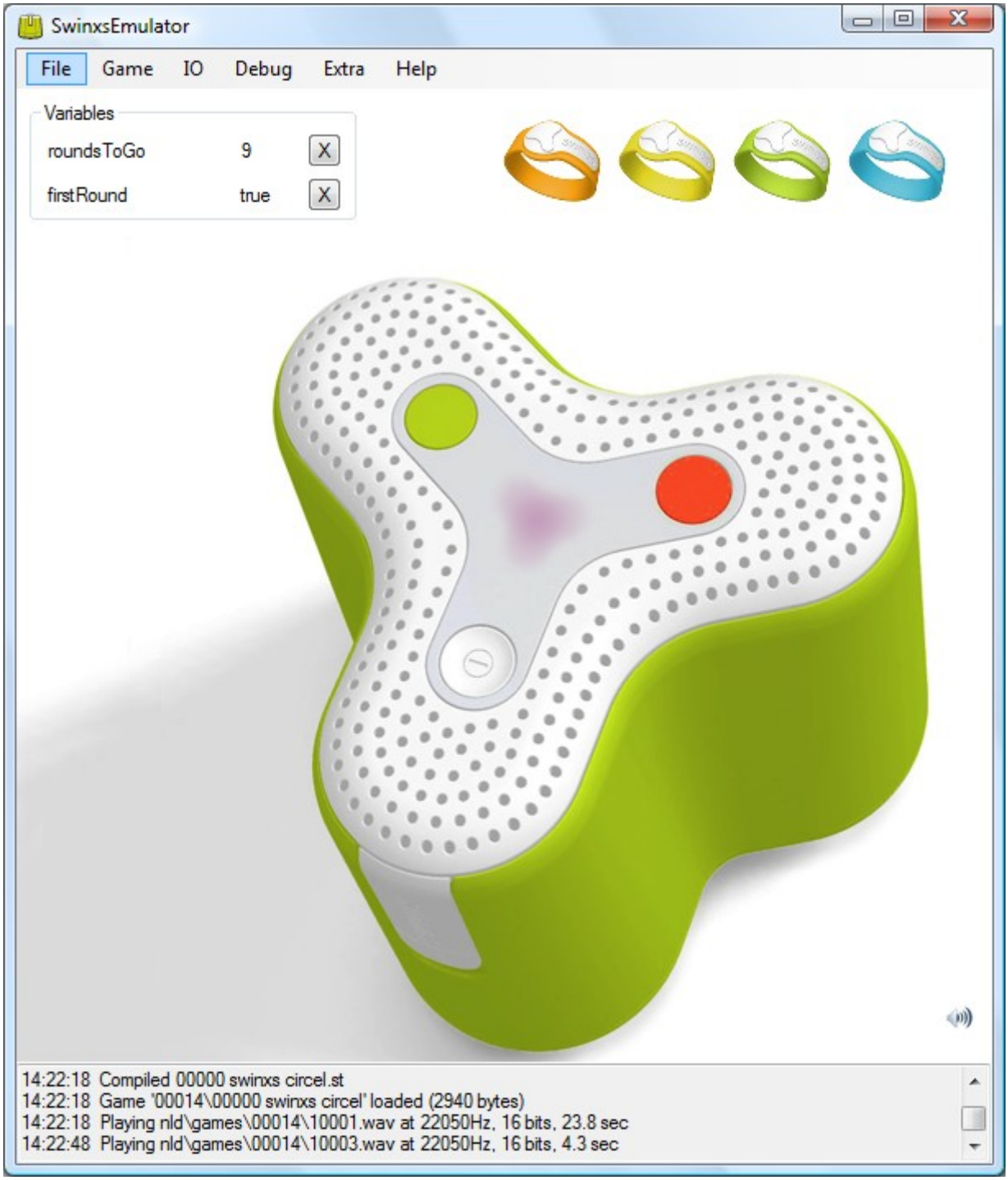
Part of the SwinxsSDK is the option to download most of the games developed by Swinxs company. Some of these games are written in SwinxsTalk which you can use as a starting point:

- 00011 – Swinxs Dash
- 00045 – Hide your XS
- 00047 – Relay
- 60001 – Animal Battle (playing cards game)
- 60002 – Lost & Found (playing cards game)

You might use “Hide your XS” as a starting point.

5. SwinxsEmulator

One of the purposes of the emulator is to compile and run SwinxsTalk and SwinxsCoreTalk games. The extension of SwinxsCoreTalk games is *.sct. For SwinxsTalk games, it is *.st. Typically, you load boot.sct, and Swinxs is running on your PC:

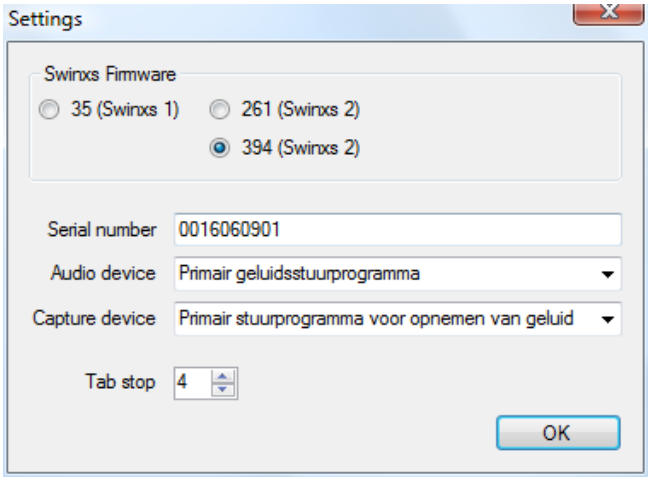


You have a few means to operate the input devices:

- Press the green, red, and white buttons to issue the button events button.*.down and button.*.short booleans. Press and hold to set the button.*.long and button.*.hold booleans.
- Swinxs will detect a XS when you press a colored XS button (at the top-right of the window). You can also drag them over Swinxs. Add additional XS-tags via the “IO” menu.
- When you want to use playing card tags, select the menu “IO”, “Show Animal Cards” or “Show Picto Cards”.
- The trigger a move event, drag Swinxs with your mouse. Shaking Swinxs will fire the shake event.
- Select “Low Power” from the “IO” menu to get Swinxs in a low power condition.
- Got tired of the Swinxs' voice? Just tick the mute checkbox.
- The emulator will use your microphone for voice recordings and the noise event. To select your audio capturing device (microphone), use the Settings dialog.

5.1 Settings

Using the “Settings” menu option, you can control the behavior of the emulator:



Normally, you run the emulator in the highest available firmware version. But if you want to see how your game runs on an older Swinxs, you can change it. See the swinxs.cabability(“..”) expression for a list a functional changes between firmware revisions.
The serial number is used mainly when you write a log file. Also your local SYSTEM.PRP will be updated to reflect your serial number.

In order to make voice samples from within the emulator, you might need to select another source as the capturing device.

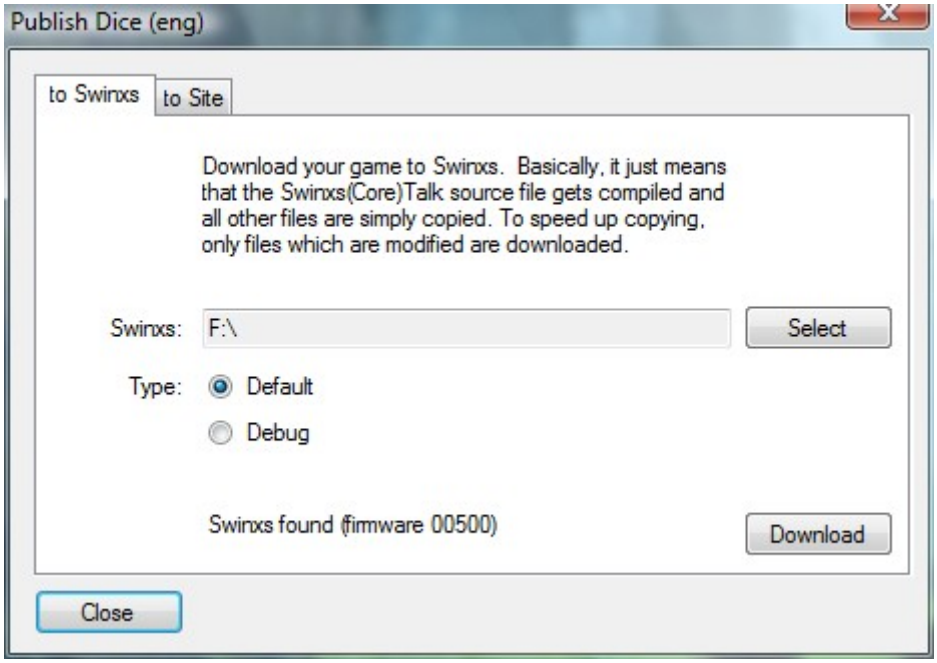
5.2 Debugging code

You have a few options to your code:

- Use game.debug.log(<variable>) statements to show the contents in the Swinxs status window at the bottom;
- Use game.debug.break statements to stop executing your code at specific spots. You can resume running your game with F5, of step through your code using F10.
- If the DEBUG constant, e.g. “if (DEBUG == 1) { … } else { … }” or “DEBUG == 1 ? … : ...”.
- View the contents of word- or boolean variables by adding watches. Just click on the “Add” button in the block labeled “variables” at the top-left and select the variable you want to follow.

5.3 Downloading your game to Swinxs

You can publish game to Swinxs. Just connect Swinxs to your PC and run your game in the emulator. Then choose File, Publish:

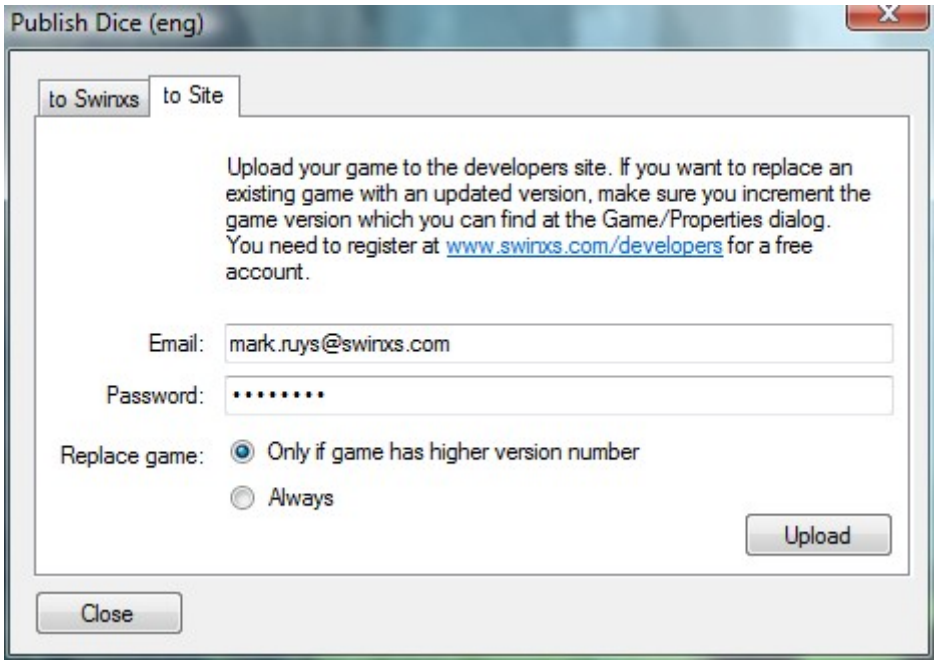

Publish Dice (eng)
to Swinxs
to Site
Download your game to Swinxs. Basically, it just means that the Swinxs(Core)Talk source file gets compiled and all other files are simply copied. To speed up copying, only files which are modified are downloaded.
Swinxs:
F:\
Select
Type:
Default
Debug
Swinxs found (firmware 00500)
Download
Close

Press “Download” to start the publishing process. All *.sct and or *.st file will be compiled to *.bin files, and all other files will be copied to your Swinxs.

If you select “Debug”, your code will be compiled in debug-mode. The constant DEBUG is defined by 1 when “Debug” is selected, by 0 otherwise.

5.4 Uploading your game to the developers site

When you've finished your game, you are encouraged to share it with others. Essentially it comes down to first register yourself as a developer at developers.swinxs.com (which you probably already did to obtain a valid game.prp file). Then choose “File”, “Publish...”, “to Site” tab:


Publish Dice (eng)
to Swinxs
to Site
Upload your game to the developers site. If you want to replace an existing game with an updated version, make sure you increment the game version which you can find at the Game/Properties dialog.
You need to register at www.swinxs.com/developers for a free account.
Email:
mark.ruys@swinxs.com
Password:
Replace game:
Only if game has higher version number
Always
Upload
Close

Fill in your Login and Password, and press “Upload”. Your game will be zipped and uploaded.

5.5 Shortcut keys

The emulator knows the following shortcut keys:

Control-G	Button green
Control-R	Button red
Control-W	Button white
F5	Start or Continue
Shift-F5	Stop
F10	Step
Shift-Control-F5	Restart (reboot Swinxs)

5.6 Text-to-speech

The emulator has build in text-to-speech. You can use this while developing a game when you do not have the final samples ready. Instead of a wav file like 00012.wav, you create a file 00012.txt. The emulator will speak out the contents of this document in English. Also, the sample is immediately saved as a wav file.

5.7 Syntax highlighting

The debugger of SwinxsEmulator does not support syntax highlighting. Syntax highlighting and command completion for SwinxsTalk files is available for the freely available text editor Notepad++. You find the necessary configuration files in %ProgramFiles%\Swinxs\SwinxsSDK\Notepad++.

1. Copy SwinxsTalk.xml and SwinxsTalk.api into %ProgramFiles%\Notepad++\plugins\APIs.
2. Open %APPDATA%\Notepad++\userDefineLang.xml, and add the <UserLang> section from the provided userDefineLang.xml. Beware that %APPDATA% and %ProgramFiles% are usually not the same folders.

When you open a SwinxsTalk (*.st) file for the first time in Notepad++, choose Syntax, SwinxsTalk to enable syntax highlighting.

6. Game development cycle

In a nutshell, follow these steps to develop a new game:

1. Design a new game conceptually.
2. Write down a game definition. You could use an existing game as a starting point: in the folder SwinxsSource, you'll find a a subfolder 'games'. Every game listed there contains an Excel sheet. The first tab is the game definition. It describes a game in terms of a final state machine.
3. Using the SwinxsEmulator, start developing your own game: Choose “Game”, “New”, “Hello World” to create a subdirectory in the games folder (SwinxsSource/games/<mygame>).
4. Start writing your code. Run your game in the SwinxsEmulator to debug it.
5. To add samples, just create a <snd_file>.txt file with the text in English. The SwinxsEmulator will use speech synthesis to generate the <snd_file>.wav for you automatically. Keep an administration of all used samples in a separate tab in your Excel sheet (see other games for a template).
6. Publish your game to your Swinxs using the SwinxsEmulator and check it work's alright.
7. Speak in your samples and replace the generated wave files in SwinxsSource/<LANG>/games/<mygame>/.
8. Upload your game to the site to share it with others.

7. Game design guidelines

This section outlines ‘good design’ guidelines for Swinxs games. By following these guidelines you’ll…
…assure some level on consistency between all Swinxs games
…avoid some of the pitfalls that were learned the hard way with the first Swinxs games.
Of course in certain specific scenarios you may have strong reasons to deviate. But mostly you're game will be more intuitive and enjoyable for Swinxs users when you follow these guidelines.

Typical game structure…

A typical structure for a Swinxs session is as follows:

1. Kids register with Swinxs;
2. They select a game;
3. Swinxs explains the game;
4. The kids play the game, where a game may consist of several rounds;
5. The game is completed - optionally Swinxs calls out the scoring;
6. Swinxs prompts the kids to play the same game again or to select another.

Note that step 1, 2 and 6 are already embedded in Swinxs’ software and are not covered in this section.
Also note that number of players is determined at step 1 and is therefor fixed per game. In other words within a game the number of (registered) players can not change. The guidelines listed in the following paragraphs are therefor targeting especially step 3, 4 & 5, the steps that define your game.

Use a very short game instruction

Most games require a ‘instruction’ phase in which Swinxs explains the game play. Keep this instruction as short as possible as kids have little patience for just listening.
If there are many rules to explain try to incorporate these rules in the game’s progression instead of explaining it all at the star (e.g. with new rounds extra rules may be explained).

Keep the dialog short

Also during the game play you should keep all dialogs very short! If needed use a longer text for the first time it is used, and a short version for subsequent rounds. So first example when calling out a player for the first round Swinxs may say:

“Red, now it's you turn to hide”

For subsequent rounds this may be shortened to:

“Red, your turn!”

Or simply:

“Red!”

Refer to players only via nouns

Always refer to players using their associated color as a noun, never as an adjective. So do not use:

“It’s Blue’s turn!”

But instead:

“Blue, your turn!”

Only using nouns also makes your dialog more compact. So do not use:

"The Blue player has won"

Instead use:

"Blue has won"

Avoid repetition

Repeating identical phrases grows old really quickly. For often repeated dialogs provide synonyms/alternative texts to keep things fresh. E.g. for rewarding a player you could define a set of phrases and pick one randomly:

"Well done!"

"Excellent!"

"Couldn't do it better myself"

Etc.

Avoid over-confirmation

Overly use of confirmations kills game play. For example rather then requiring the start of each new round to be confirmed with a press on the green key, let the game progress automatically or with a well-chosen time-out.

Provide sense of progress for games that have several rounds

For games that take longer to play, consider some points in the game where you give some feedback on 'where' they are, e.g.:

"Just two more rounds to go…"

Common behavior of the Red and the Green key.

Swinxs has just three keys. The Red and Green key are available for your game. However keep the following behavior in mind…

- Green typically is the key that allows the kids to 'pace' the game play. During most Swinxs dialogs (especially during relatively long instructions) a short press on the Green key should skip the dialog to the next step in the interaction. This allows kids that already know the game to directly jump to the fun.
- During dialogs the Red key behaves as the opposite of the Green key. Where the green key skips the dialog, a short press on the Red key will let Swinxs repeat the current dialog. This is a handy trick in case the kids missed or did not understand some of Swinxs' text.
  Red's other main main use is as an emergency exit from the current game: a long press (i.e. pressed for at least 2 seconds) will trigger the 'stop this game and select another?' dialog.

You can decide not to support the short press skip/repeat behavior, but only if you have a clear rationale, for example if it interferes with your specific game play. The 'long press on Red' emergency exit however should always be supported.

Don't use the White key

White can not be used for game play. The white key is intended to allow kids to toggle the volume level at any moment during a game. A long press will switch Swinxs on/off.

Test your game before implementation

Before implementation simulate your game script: while someone plays the role of Swinxs (using your script) kids can already experience the game play.